基于 QWorker 的多线程编程

作者: swish
版本: 1.1
末次修订日期: 2015-4-2

QDAC 开发组 swish 作品
官方网站： http://www.qdac.cc
官方 QQ 群： 250530692
开源网址： http://sourceforge.net/projects/qdac3
QWorker 专题： http://www.qdac.cc/?p=191

首先欢迎各个使用 QDAC 相关的各项组件！尤其感谢各位 QDAC 官方群的朋友的集思广义、测试和赞助支持，有你们的支持，才有 QDAC 的不断进步。

QDAC 是一个免费的开源项目，您不需要为使用本项目的内容支付任何费用。如果您觉得本源码对您有帮助，愿意赞助本项目的发展（非强制），以使作者不为生活所迫，有更多的精力为您呈现更好的作品，可以赞助作者。
QDAC 允许您自由的复制、分发、修改本源码，但您的修改应该反馈给作者，并允许作者在必要时，合并到本项目中以供使用，合并后的源码同样遵循 QDAC 版权声明限制。

您的产品的关于中，应包含以下的版本声明:

本产品使用的 XXXX 来自 QDAC 项目，版权归作者所有，官方网站：www.qdac.cc。

【注意】

作者对本项目的品质不提供任何担保，您在使用本项目的源码之前，请确认这一点。QDAC 开发组不为由此给您及您客户造成的任何损失承担任何法律及道义责任。

【技术支持】

如有技术问题，您可以加入 QDAC 官方 QQ 群 250530692 共同探讨，但请注意这是一个技术群，偶尔风花雪月没什么关系，但不要刷屏造成大家的困扰，同时，不要进行任何人身语言攻击，否则将被 T 出群。
本项目目前没有提供任何帮助文件，所有的函数等我尽量在源码中加入足够的注释，以便大家了解。

如有其它问题，可以发邮件到 109867294@qq.com。

扫描下面的二维码加入 QDAC 官方 QQ 群：

如果需要单独培训或定制，可以作者单独联系，费用根据实际情况另行计算。

【赞助】

本项目接受各位热心朋友的赞助，赞助方式：

支付宝：

guansonghuan@sina.com 姓名：管耸寰

建设银行：

户名：管耸寰

账号：4367 4209 4324 0179 731

开户行：建设银行长春团风储蓄所

目 录

第一章 引言

QWorker 是 QDAC 项目带给大家的一套多线程并行编程框架。它基于作业的视角来规划作业多线程编程，简化多线程编程的步骤和方法和交互手法。

首先，我们要明白我们为什么需要多线程并行编程？

我们之所以使用多线程编程，一般目的不外乎下面两个：

1、避免程序界面假死带来的恶劣用户体验。

2、充分利用现代计算机的处理资源，来加快业务的处理速度。

当然，您可能有更好更多的理由来做多线程编程，但上面的这两个理由对于一般的人来说足够了。

接下来，一个新的问题来了，我们为什么要什么 QWorker 而不是系统自带的多线程框架来编程？

这个理由实际上很简单，使用 QWorker 会进一步简化你的编程逻辑的设计，使你更专注于业务流程的规划和实现，而不去管理线程池及处理资源的调度问题。

那么，什么是作业？

在 QWorker 中，所谓的作业就是一个逻辑上的业务处理单元。我们要实现一个复杂的任务时，可以将其分解成一到多个小的任务，每个任务由一个函数来管理，这个函数我们就可以将其理解为一项作业，而这些函数的组合就构成了一个作业分组。作业我们可以让其运行在主线程或者后台线程，注意一点，主线程不适合进行长时间作业，那样会造成主线程阻塞，造成假死的现象，影响用户体验。

现在看看，QWorker 为我们提供了什么呢？

1、一个跨平台的异步执行体系

一旦我们通过 Workers.Post 直接来投寄一个异步作业，这个作业将在后台被计划执行。

2、一个跨平台的计划任务框架

通过 At 函数，可以满足作业定时执行的需求。

3、一个高精度的后台线程定时器

通过 Post 时设置重复间隔，可以简单的生成一个定时重复作业，实现后台线程定时器的效果。

4、一个松散耦合的业务框架

通过信号建立不同模块的不同单元的联系，可以很好的实现模块间的松散耦合和自动触发。

5、一个流程管理框架

通过作业组（TQJobGroup）可以将多个作业管理在一起，然后串行或并行执行，并在全部作业执行完成时，得到相应的事件通知，当然也可以等待全部作业执行完成。

6、一个并行操作引擎

通过 Workers.For 可以并行循环执行同一个过程，而直接 Post 到后台的作业，也会被计划成并行执行。

7、一个线程池

QWorker 提供了一个自动管理线程生命周期的线程池。当然如上所述，它的功能远不止于此。

我们接下来，从现实的逻辑上来对比以加深 QWorker 体系的理解。我们先将其中的触角分配一下：

作业（JOB） - 用户交付要完成的任务

工作者（Worker）- 完成任务的工人

作业管理器（Workers）- 包工头

首先，包工头会雇佣少数固定工人以完成基本的工作，它的数量，在 QWorker 中，由 Workers.MinWorkers 决定，默认值为 2，不能小于 1。没有这些工人，一旦有作业过来，包工头得现招人，显然很浪费时间，而如果始终养着一帮工人，对于包工头来说，显然投资（资源占用）太大。一旦作业多了，忙不过来怎么办，那就需要招聘新工人来解决了所有工人的数量由 Workers.MaxWorkers 属性来决定，默认是 CPU 核心数*2+1。但长期养着那么多临时工是不可能的，但立即解雇显然也不好（万一后面紧接着有新作业到来还得现雇），所以包工头会在一段时间没活后，将尽量解雇工人直到达到 Workers.MinWorkers 指定的限制，以减少投资。这个时间，由 Workers.FireTimeout 来决定，默认是 15 秒。

当作业被用户交付给包工头后，根据作业的不同，需要包工头在不同的时间点安排工人去执行。到了执行的时候，包工头会从现在雇佣的工人中，找有没有空闲的工人，如果有，直接交付处理，如果没有，则雇佣新的工人来完成作业。

在后面的章节，我们将进一步分析和了解如何使用 QWorker 进行多线程编程。

第二章 不同的作业视角

前一章我们说了，QWorker 是基于作业的视角来考虑的，所以，这一章，我们试图从不同的视解来解读下作业的分类，古语说“横看成岭侧成峰，远近高低各不同”，我们也横七竖八的切分下看看。

1、从作业的运行线程环境来分

我们知道，许多东西如窗体上的控件，都不是线程安全的，如果我们在后台线程中直接访问它，会发生意想不到的错误。所以，如果作业需要访问窗体上的东西，那么千万要记得将作业的 ARunInMainThread 参数设置为 True 。从这一点上分析，我们可以将作业分为主线程作业和后台线程作业。

主线程作业

主线程作业适合于做一些用户界面相关的交互操作，主要是因为用户界面上的各个控件，很少有是线程安全的。

- 后台线程作业

后台线程作业适合于做实际的数据获取及处理工作，在处理过程中，如果需要与界面交互，就可以直接投递一个主线程作业来处理。在后台线程作业中，我们许多时候，需要考虑多线程同步的问题，希望大家在设计程序时一定要注意这一点。

2、按作业的执行时间点划分

我们的作业投递给 Workers 这个包工头后，我们可能希望立即安排工人去完成作业，也可能希望过一会再去完成作业，甚至我们可能希望在我们给包工头一个眼神后，包工头再安排工人去完成作业。所以，从作业的执行时间点上，我们将作业分为立即执行、延迟执行和等待信号执行三种。

- 立即执行

我们将作业投递（Post）给 Workers 这个包工头后，它会尽量立即安排工人去执行作业（LookupIdleWorker）。注意一点，这个作业如果要求运行在主线程，而你的投递操作也在主线程，那么它必需等待主线程进入消息处理循环时才能得到处理。同时，如果没有空闲的工人可用，你的作业也会被放到队列里等待有空闲工人时才能被处理。

- 延迟执行

我们将作业投递（Delay/At）给 Workers 这个包工头，要求它延迟一段时间再去执行（比如说 2 秒后执行）。注意一点，QWorker 里，时间的最小单位是 0.1ms，所以，如果 1 秒对应的值是 10000，您可以用 Q1Second 来代替。

- 等待信号

在看谍战电影时，我们许多时候，看见谍报人员在阳台上放盆花，一旦花盆没了，那就说明出事了，而这就是所谓的信号的一种。我们可以将作业投递给 Workers 这个包工头时，告诉它等待（Wait）我们的特定信号。一旦我们触发了相应的信号（花盆没了:)），包工头会立即安排工人去执行指定的作业。

因为信号的发出和执行的作业是完全分离的，所以，信号的触发不需要知道是否有人响应，而信号的响应者不需要知道啥时候自己该去执行，从而实现了业务逻辑的松散耦合。

3、按作业的执行次数分

一个作业，有可能只执行一次，有可能需要重复执行多次。从这个视角上来分，我们将作业分为单次作业和重复作业两种。顾名思义，单次作业在我们投递后，只会被调度执行一次，重复作业则相反，它将至少执行一次。

单次作业的触发，我们一般调用 Workers.Post、Workers.Delay、Workers.At 函数来触发，不过它的重复间隔参数（AInterval）被设置为 0，如果我们在投递时，设置其重复间隔不为 0，则作业会被重复的调度执行。

对于等待特定信号触发的作业（Workers.Wait），我们一般认为它是重复作业，只是重复的频率是不定的，取决于信号的触发次数。

4、从作业的触发方式分

一个作业啥时候被交付给工人执行，从这个视角，我们对作业来观察，会发现有三个不同的分类手工触发、定时触发和信号触发。

- 手工触发

这类作业一般是由程序员在需要时直接在投递（Post）一个后台作业给包工头，包工头会立即安排作业的执行。

- 定时触发

定时触发的作业，要求作业在特定的时间点运行，它一般由 Delay、At 函数投递给包工头，由包工头内部在到达指定的时间后，触发执行。

- 信号触发

就象前面说的，信号触发的作业是通过 Wait 函数投递给包工头的。而在另外的地方，通过 Signal 函数触发信号后，由包工头调度执行。

与前面两种方式相比，信号触发类型的作业步骤稍显多一些：

- 信号注册

调用 Workers.RegisterSignal 来注册一个信号，然后会返回一个用来唯一标记这个名称信号的 ID。

- 作业投递

调用 Workers.Wait 函数告诉 QWorker 自己要等待的信号名称或 ID，推荐用 ID，开销更小一点点，但用信号当然也可以，只不过触发时，需要先遍历所有信号找到相应的信号 ID，然后执行触发过程。

- 信号触发

在需要执行相关操作时，调用 Workers.Signal 函数触发信号，同样推荐使用 ID。在上一步投递的信号响应作业，会自动被触发，至此完整的信号触发过程完成。

上面列出了作业自己从不同视角对作业的分析。诚然，你完全可以从另外的角度去分析作业的不同分类，欢迎进一步的补充和分析。

第三章 第一个作业

要使用 QWorker 进行后台线程作业，我们要做的第一件事，就是当然就是要引用 QWorker 这个单元，在 C++ Builder 中，要包含 qworker.hpp。

【Delphi】

```
uses qworker;
```

【C++ Builder】

```
#include "qworker.hpp"
```

在 C++ Builder 中，我们还需要以下两步之一：

（1）、将 qworker 相关的 .pas 加入到当前工程中，这样 C++ Builder 就会自动编译生成 qworker.hpp，并在链接时将 qworker.obj 链接到程序中。

（2）、先建立一个临时的工程，将 qworker 相关的 .pas 加入后编译生成的 *.hpp 复制到 C++ Builder 的 Include 目录下，将*.obj 复制到 Lib 目录下，这样以后只需要直接包含 qworker.hpp 就可以了。

做好上面的准备工作后，我们就可以做一个最简单的作业了。好吧，我们的作业只干一件事，显示一个经典的的“Hello，world!”。

首先添加一个主线程作业处理函数 DoHelloworld，这个函数之所以要求运行在主线程，是因为用 ShowMessage 是 VCL 函数，它不是线程安全的，所以要运行在主线程中。然后我们在窗体上添加一个按钮 Button1，然后设置其 OnClick 事件来触发作业处理函数。完成后的代码如下：

【Delphi】

```
unit Unit1;
 interface
 uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs,qworker, Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
```

```
    procedure DoHelloworld(AJob:PQJob);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
Workers.Post(DoHelloworld,nil,True);
end;

procedure TForm1.DoHelloworld(AJob: PQJob);
begin
ShowMessage('Hello,world');
end;
end.
```

【C++ Builder】

（1）头文件：Unit1.h

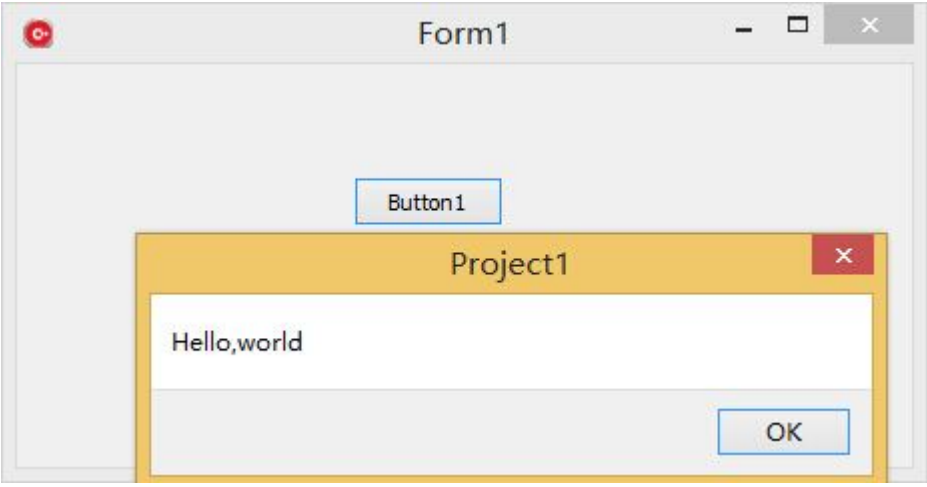
```
//---------------------------------------------------------------------------

#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <System.Classes.hpp>
#include <Vcl.Controls.hpp>
#include <Vcl.StdCtrls.hpp>
#include <Vcl.Forms.hpp>
#include "qworker.hpp"
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published: // IDE-managed Components
TButton *Button1;
```

```
void __fastcall Button1Click(TObject *Sender);
private: // User declarations
void __fastcall DoHelloworld(PQJob AJob);
public: // User declarations
__fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------

#endif
```

（2）主文件：Unit1.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::DoHelloworld(PQJob AJob)
{
ShowMessage("Hello,world");
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Workers->Post(DoHelloworld,NULL,true);
}

//---------------------------------------------------------------------------
```

看看程序的运行效果：



好了，效果看到了。我们看看刚才用到的 Post 函数的声明：

【Delphi】

```
function Post(AProc: TQJobProc; AData: Pointer;ARunInMainThread: Boolean = False;AFreeType:
TQJobDataFreeType = jdfFreeByUser): IntPtr; overload;
```

【C++ Builder】

```
NativeInt __fastcall Post(TQJobProc AProc, void * AData, bool ARunInMainThread = false,
TQJobDataFreeType AFreeType = (TQJobDataFreeType)(0x0))/* overload */;
```

这个函数的作业是投寄一个需要立即在后台执行的作业，如果成功，返回一个作业的句柄实例，如果失败，返回 0。其参数含义如下：

AProc ：要执行的作业处理函数

AData ：作业附加的用户数据指针，其具体含义由作业处理函数自行解析，可以用来传递作业需要的参数

ARunInMainThread：作业是否要求在主线程中执行，如果作业需要在主线程执行，则设置为 true，如果不需要，则设置为 false

好了，这个最经典的 HelloWorld 就此搞定，是不是觉得很简单？好复杂？！算了，编程不适合你，开玩笑:)

第四章 传递参数给作业

在上一节中，我们看到这个示例实在是太简单了，实际在作业处理过程中，稍微复杂一点的作业，我们都需要传递参数给作业的处理函数。那么怎么传递参数给作业呢？

在 Post 函数的声明中，我们看到有一个 AData 参数，是一个无类型的指针，我们实际上就是通过它来作业传递参数。我们知道，无类型指针实际上是一个整数来记录地址的值，在 32 位程序中，它是 32 位的，在 64 位程序中，它是 64 位的，那么，我们实际上，对于小于等于 32 的内容，是可以直接传值的，这样做的好处是不需要额外的释放步骤，而 64 位的内容则需要视我们的具体程序环境而定，具体可参考下表：

数据类型		32 位	64 位
Delphi	C++ Builder		
AnsiChar	char	√	√
BYTE	unsigned char	√	√
WideChar	wchar_t	√	√
Shortint	char	√	√
WORD	unsigned short	√	√
Smallint	short	√	√
Integer	int	√	√
Cardinal	unsigned int	√	√
Longint	long	√	√
Int64	__int64	×	√
Single	float	√	√
Double	double	×	√
Currency	Currency	×	√
TDateTime	TDateTime	×	√

当然，这些可以直接传值的类型，我们也需要做一些简单的技巧性处理，如传递一个字符时，Delphi 中就可能需要用到 Ord 函数传它的内码，而传递一个整数类型的值时，我们就需要强制转换一个，Pointer（值）即可，而传递浮点值时，则需要更复杂一点的转换，您可以用 absolute 语法，也可以使用取地址转换的办法来完成。如下所示：

```
//整数类型
Pointer(100);
//浮点类型-absolute
var
  f:Single;
  i:Integer absolute f;
```

```
begin
...
f:=1.2;
...
Pointer(i);
...
end;
//浮点类型-强转

Pointer(PInteger(@f)^);
```

当然，到了作业中相应的需要将 Pointer 反转换来就可以了，具体就不再缀述，省得大家认为我在骗字数。

上面的简单类型传递已经搞定了，那么复杂的类型传递我们怎么传呢？大家可能注意到了 Post 函数还有一个参数叫 AFreeType，它指定了 AData 指针关联的对象如何去释放。QWorker 默认提供了下面几种释放方式：

- jdfFreeByUser ：用户自己负责，QWorker 可以当甩手掌握的
- jdfFreeAsObject：传递的是一个对象指针，QWorker 在作业生命周期结束时，调用对象的析构函数释放
- jdfFreeAsSimpleRecord：传递的是一个简单记录类型，QWorker 在作业生命周期结束时，直接调用 Dispose(Pointer) 来释放，注意不要传递包含字符串类型/接口/动态数组等类型的成员，否则会造成内存泄露
- jdfFreeAsInterface：传递的是一个接口，QWorker 会自动调用 AddRef 和 Release 来维护其生命周期
- jdfFreeAsC1 ~ jdfFreeAsC6：传递的是一个用户自定义的内存释放方式，由用户自行指定 Workers.OnCustomFreeData 事件的响应函数来处理

看起来挺多的，但实际的业务需要可能更复杂，那么我们能不能进一步简化呢？万能的神龙，变变变~~~TQJobExtData 登场了，AData 参数传递的是 TQJobExtData 这个类型的一个实例，然后由它提供了一层封装。当然了，因为 TQJobExtData 是一个类，所以 AFreeType 必需手动指定为 jdfFreeAsObject。我们看这个类的声明：

```
TQJobExtData = class
  private
    function GetAsBoolean: Boolean;
    function GetAsDouble: Double;
    function GetAsInteger: Integer;
    function GetAsString: QStringW;
    procedure SetAsBoolean(const Value: Boolean);
    procedure SetAsDouble(const Value: Double);
    procedure SetAsInteger(const Value: Integer);
    procedure SetAsString(const Value: QStringW);
```

```
    function GetAsDateTime: TDateTime;
    procedure SetAsDateTime(const Value: TDateTime);
    function GetAsInt64: Int64;
    procedure SetAsInt64(const Value: Int64);
  protected
    FOrigin: Pointer;
    FOnFree: TQExtFreeEvent;
{$IFDEF UNICODE}
    FOnFreeA: TQExtFreeEventA;
{$ENDIF}
    procedure DoFreeAsString(AData: Pointer);
    procedure DoSimpleTypeFree(AData: Pointer);
{$IFNDEF NEXTGEN}
    function GetAsAnsiString: AnsiString;
    procedure SetAsAnsiString(const Value: AnsiString);
    procedure DoFreeAsAnsiString(AData: Pointer);
{$ENDIF}
  public
    constructor Create(AData: Pointer; AOnFree: TQExtFreeEvent); overload;
    constructor Create(AOnInit: TQExtInitEvent;
      AOnFree: TQExtFreeEvent); overload;
{$IFDEF UNICODE}
    constructor Create(AData: Pointer; AOnFree: TQExtFreeEventA); overload;
    constructor Create(AOnInit: TQExtInitEventA;
      AOnFree: TQExtFreeEventA); overload;
{$ENDIF}
    constructor Create(const Value: Int64); overload;
    constructor Create(const Value: Integer); overload;
    constructor Create(const Value: Boolean); overload;
    constructor Create(const Value: Double); overload;
    constructor CreateAsDateTime(const Value: TDateTime); overload;
    constructor Create(const S: QStringW); overload;
{$IFNDEF NEXTGEN}
    constructor Create(const S: AnsiString); overload;
{$ENDIF}
    destructor Destroy; override;
    property Origin: Pointer read FOrigin;
    property AsString: QStringW read GetAsString write SetAsString;
{$IFNDEF NEXTGEN}
    property AsAnsiString: AnsiString read GetAsAnsiString
      write SetAsAnsiString;
{$ENDIF}
    property AsInteger: Integer read GetAsInteger write SetAsInteger;
    property AsInt64: Int64 read GetAsInt64 write SetAsInt64;
```

```
    property AsFloat: Double read GetAsDouble write SetAsDouble;
    property AsBoolean: Boolean read GetAsBoolean write SetAsBoolean;
    property AsDateTime: TDateTime read GetAsDateTime write SetAsDateTime;
end;
```

它在这里提供了常用的类型的构造函数封装来简化操作，比如，我们要在上一节的例子中，传递一个要显示的字符串给 DoHelloworld 这个作业函数，我们就直接用下面的代码即可：

【Delphi】

```
Workers.Post(DoHelloworld,TQJobExtData.Create('字符串内容'),True,jdfFreeAsObject);
```

【C++ Builder】

```
Workers->Post(DoHelloworld,new TQJobExtData('字符串内容'),true,jdfFreeAsObject);
```

那么，我们在作业中，如何访问参数的值呢？大家可能已经注意到了对应的 AsXXX 属性，根据对应的类型，使用它即可直接访问了。当然，如果是你自定义的结构体或对象，那么只好老老实实的访问 Origin 指针来得到原始的内容了。

接下来的问题是：如果我们要传复杂的结构体给作业时怎么办？一个程序用到的结构体可能远不止 6 种，jdfFreeAsC1~jdfFreeAsC6 显然是无法满足需要的。在刚才的 TQJobExtData 声明中，大家可以看到，其包含了一个带有自定义释放函数的形式，实际上只要传递释放内存的代码给它就搞定了。

好了，看起来一切都很完美。传参数的问题解决了，也不需要做太多额外的工作。但我还是想补充一点点，那就是实际上，如果需要，你可以通过 TQMsgPack 或者 TList 一类的对象来传递参数，反正只要指定了 jdfFreeAsObject，在作业生命周期结束时，QWorker 都会帮你自动释放它。所以，本章的内容，只是给的参考，并不代表必需这么做，条条大路通罗马，何必单恋一枝花~~~，词有点串了:)，本章内容结束。

第五章 后台线程与用户界面交互

我们知道，后台作业不能直接访问和操作用户界面元素。实际上，所谓的直接访问并不是绝对的，但属性这个东西你读时，可能执行的是一个读函数，它是否内部进行了一些不安全的写操作，有时候是叫不准的，那么，显然不在后台作业中访问更安全。

那么，我们如果需要和前面的用户界面交互该怎么做？

在进一步讨论之前，我们首先要明白和主线程进行交互，实际上存在同步和异步两种情况的：

（1）、异步的交互适合通知类型的情况，它只是将要交互的内容发往主线程，由主线程根据相应的参数进行处理。

（2）、同步的交互适合于需要等待主线程处理结果的情况，它需要将交互的内容发往主线程后，由主线程返回结果给自己。

那么，同步的异步真的是绝对的吗？当然不是，实际上同步的操作可以很容易的转换为异步的操作。比如，我们异步通知主线程后，然后由主线程执行操作完后，再触发一个后台作业的异步执行，那效果和同步实际上并没有什么本质的区别。只是同步的编程模型在许多时候，更容易理解，代码的可读性更好。但反过来，同步的编程模型必需等待结果返回，在此过程中，当前线程必需等待而不能处理其它任务，从而影响处理效率。所以，我们一般推荐都是异步来进行处理的。

在 QWorker 中，我们推荐的基本处理流程是：

（1）、后台作业投寄主线程作业后退出；

（2）、主线程作业完成后，触发新的后台作业后退出；

（3）、新的后台作业进行进一步处理；

在这些步骤中，参数的传递可以通过 AData 成员传递，也可以通过将作业过程写成一个类的成员，然后直接访问类的成员变量来解决（类似于下面的分组作业处理）。

但是，作为一个有良心的开发者，我还是希望你了解的更多，所以我们进一步来阐述这一问题。

上面的步骤如果多的话，你可能觉得自己处理有点麻烦，所以 QWorker 为你提供了作业分组对象：TQJobGroup。你只需要创建一个顺序执行的作业分组，然后将步骤依次加入到分组，然后调用 TQJobGroup.Run 就好了，TQJobGroup 会依次执行作业，而不需要你再手工投寄。当然了，作业的公共参数也就不能通

过作业的 AData 成员传递了，而应该将其赋值给 TQJobGroup 的 Tag 标签，以便所有的作业共享，或者你创建一个类，将所有的公共参数和作业处理函数都做为这个类的成员，这样子就两全齐美了，不用受标签的 Tag 指针的限制了。大概示意如下：

【Delphi】

```
TMyGroupJobs=class
protected
  //自己的变量声明，然后声明作业分组对象
  FGroup:TQJobGroup;
  //声明作业处理步骤函数
  procedure DoStep_1(AJob:PQJob);
  ...
  procedure DoStep_n(AJob:PQJob);
  //用于响应 FGroup.AfterDone 事件，在所有作业完成时自动释放
  proceduer DoJobsDone(ASender:TObject);
public
  constructor Create;overload;
  destructor Destroy;override;
end;
...
constructor TMyGroupJobs.Create;
begin
inherited;
...
FGroup:=TQJobGroup.Create(True);
FGroup.AfterDone:=DoJobsDone;
FGroup.Prepare;
FGroup.Add(DoStep_1..
FGroup.Run;
...
end;

constructor TMyGroupJobs.Destory;
begin
FGroup.Cancel;
FGroup.Free;
inherited;
end;

proceduer TMyGroupJobs.DoJobsDone(ASender:TObject);
begin
FreeObject(Self);
```

```
end;
...
AJobs:=TMyGroupJobs.Create;


....
```

【C++ Builder】

```
class TMyGroupJobs
{
protected:
   //自己的变量声明，然后声明作业分组对象
   TQJobGroup *FGroup;
   //声明作业处理步骤函数
   void __fastcall DoStep_1(PQJob AJob);
   ...
   void __fastcall DoStep_n(PQJob AJob);
   //用于响应 FGroup.AfterDone 事件，在所有作业完成时自动释放
   void __fastcall DoJobsDone(TObject *ASender)
   {
   delete self;
    }
public:
   __fastcall TMyGroupJobs()
   {
   ...
   FGroup=new TQJobGroup(true);
   FGroup->AfterDone=DoJobsDone;
   FGroup->Prepare();
   FGroup->Add(DoStep_1,...
   FGroup->Run();
   }
   __fastcall ~TMyGroupJobs()
   {
   ...
   FGroup->Cancel();
   delete FGroup;
   }
};
...
AJobs=new TMyGroupJobs;


....
```

只需要将在主线程中运行的作业投递到主线程去执行就好了，剩下的就是等待它完成了。什么，内存泄露？AJobs 在你保存下，退出前自己释放下不就好了，这个不需要我告诉你吧。

好了，虽然说的是 QWorker，但我不介意提一下原生的方式：

（1）、PostMessage 发送通知消息到主线程，然后在主线程中响应消息处理。

（2）、SendMessage 发送通知到主线程，然后等待主线程中完成处理并返回结果。

无论 PostMessage 还是 SendMessage 函数，它们都是线程安全的，可以放心的投递而不用担心。唯一需要担心的是 PostMessage 的消息在特定的情况下，可能会得不到执行的机会（窗口在处理该消息之前被重建或销毁，原来的句柄就无效了）。当然，本着一切自愿，风险自担的原则，请大家放心使用。对了，提示下 FMX 平台下没有 PostMessage，但我们为您提供了一个模拟函数，参考主题：在 FMX 中实现 PostMessage 的方法，这里就不啰嗦了。

【注】QWorker 实际上信号作业也可以在主线程和后台线程之间调度作业完成交互，参数的共享也只需要象前述一样解决就好。

第六章 清理现场

有人雇佣了一些工人干活，但不幸的是，这个人在所有作业完成前，消失在那蓝色星球中的人海找不到了，于是，悲剧的工人在找他要工钱时，发现已经无法找到人了，于是工人暴怒，进行疯狂的破坏，结果就是异常发生了。为了避免工人的暴动，雇主应该在消失之前，清理作业，与工人进行结算，这样才能建设和谐社会嘛。

QWorker 中的作业在提交给 Workers 这个包工头后，作业肯定会访问一些对象、记录或普通的变量，在这些东西失效前，肯定要对相关的作业进行清理，以避免 AV 错误。在 QWorker 中，提供了 Clear 系列函数来完成此项操作。

```
/// <summary>清除所有作业</summary>

procedure Clear; overload;

/// <summary>清除一个对象相关的所有作业</summary>

/// <param name="AObject">要释放的作业处理过程关联对象</param>

/// <param name="AMaxTimes">最多清除的数量，如果<0，则全清</param>

/// <returns>返回实际清除的作业数量</returns>

/// <remarks>一个对象如果计划了作业，则在自己释放前应调用本函数以清除关联的作业，

/// 否则，未完成的作业可能会触发异常。</remarks>

function Clear(AObject: Pointer; AMaxTimes: Integer = -1): Integer;

  overload;

/// <summary>清除所有投寄的指定过程作业</summary>

/// <param name="AProc">要清除的作业执行过程</param>

/// <param name="AData">要清除的作业附加数据指针地址，如果值为 Pointer(-1),

/// 则清除所有的相关过程，否则，只清除附加数据地址一致的过程</param>

/// <param name="AMaxTimes">最多清除的数量，如果<0，则全清</param>
```

```
/// <returns>返回实际清除的作业数量</returns>

function Clear(AProc: TQJobProc; AData: Pointer; AMaxTimes: Integer = -1)

  : Integer; overload;

/// <summary>清除指定信号关联的所有作业</summary>

/// <param name="ASingalId">要清除的信号 ID</param>

/// <returns>返回实际清除的作业数量</returns>

function Clear(ASignalName: QStringW): Integer; overload;

/// <summary>清除指定信号关联的所有作业</summary>

/// <param name="ASingalId">要清除的信号 ID</param>

/// <returns>返回实际清除的作业数量</returns>

function Clear(ASignalId: Integer): Integer; overload;

/// <summary>清除指定句柄对应的作业</summary>

/// <param name="ASingalId">要清除的作业句柄</param>

/// <returns>返回实际清除的作业数量</returns>

procedure ClearSingleJob(AHandle: IntPtr); overload;

/// <summary>清除指定的句柄列表中对应的作业</summary>

/// <param name="AHandles">由 Post/At 等投递函数返回的句柄列表</param>

/// <parma name="ACount">AHandles 对应的句柄个数</param>

/// <returns>返回实际清除的作业数量</returns>


function ClearJobs(AHandles: PIntPtr; ACount: Integer): Integer; overload;
```

通过函数的注释，我相信大家已经明白各个清理函数都是干嘛用的了，用的时候，直接调用 Workers.ClearXXX(…) 执行清理过程就好。

如果是分组作业，那么我们该如何怎么办？ TQJobGroup 提供了一个 Cancel 函数来完成这项工作。Cancel 函数将取消后续作业的执行，注意其中的 AWaitDone 参数要慎用，如果在分组作业中的作业执行 Cancel 时，必需传递为 False，否则会死等自己结束而不可得。至于其它场合，使用默认值即可。

总之，一个稳定可靠的程序，及时清理不需要的资源是一个良好的工作习惯。

第七章 线程定时器

在实际编程环境中，我们常需要定时执行一项任务，比如每隔 1 小时，执行一次同步操作，而这些操作如果放到主线程中使用普通的 TTimer 执行，由于需要占用主线程的资源，可能会造成程序明显的卡顿。这时候，我们就需要使用后台定时器来完成这一操作（当然 QWorker 也支持主线程中的定时作业，此时就相当于你在窗口上放置了一个 TTimer 控件）。

QWorker 提供了三种不同的定时方式“

1、一个最普通的定时器，通过 Post 投寄一个带有时间间隔的作业，我们来看对应的 Post 函数的参数（全局函数和匿名函数版本略，下同）：

```
/// <summary>投寄一个后台定时开始的作业</summary>

/// <param name="AProc">要执行的作业过程</param>

/// <param name="AInterval">要定时执行的作业时间间隔，单位为 0.1ms，如要间隔 1 秒，则值为 10000</param>

/// <param name="AData">作业附加的用户数据指针</param>

/// <param name="ARunInMainThread">作业要求在主线程中执行</param>

/// <param name="AFreeType">附加数据指针释放方式</param>

/// <returns>成功投寄返回句柄，否则返回 0</returns>

function Post(AProc: TQJobProc; AInterval: Int64; AData: Pointer;

  ARunInMainThread: Boolean = False;


  AFreeType: TQJobDataFreeType = jdfFreeByUser): IntPtr; overload;
```

在 Post 时，我们通过 AInterval 参数，指定一个时间间隔就可以达到定时重复作业的目的了。比如下面的代码，每秒会在 IDE 的 Events 调试窗口输出当前时间：

【Delphi】

```
procedure TForm1.DoTheadTimer(AJob: PQJob);

begin

OutputDebugString(PChar(FormatDateTime('yyyy-mm-dd hh:nn:ss',Now)));

end;



procedure TForm1.FormCreate(Sender: TObject);

begin

Workers.Post(DoTheadTimer,1*Q1Second,nil);


end;
```

【C++ Builder】

```
// 头文件部分省略

__fastcall TForm1::TForm1(TComponent* Owner)

: TForm(Owner)

{

Workers->Post(DoThreadTimer,1*Q1Second,NULL);

}

//---------------------------------------------------------------------------

void __fastcall TForm1::DoThreadTimer(PQJob AJob)

{

OutputDebugString(FormatDateTime("yyyy-mm-dd hh:nn:ss",Now()).c_str());


}
```

很简单吧！

【注意】 这个定时器不会管你作业是否处理完成，按规定的间隔定时每秒触发 1 次，如果你长时间做业超时，会造成作业累积，后果可能会~~~~

2、使用 Delay 设置自由定时器

Delay 与 Post 不同，Delay 实际上是一个延迟指定时间执行某一操作的一种作业方式。是一个延迟定时器。由于 QWorker 允许在作业中投寄新作业，因此，在这个作业完成时，再次投寄作业就可以达到重复的目的。这样实现的定时器与第一种方式不同，它实际上类似于我们使用 TTimer 时，在其 OnTimer 事件中先通过设置 Enabled 为 False 禁用 Timer，然后在处理完成后再设置 Enabled 为 True 启用定时器的效果。

我们先看下 Delay 函数的声明：

```
  /// <summary>投寄一个延迟开始的作业</summary>
   /// <param name="AProc">要执行的作业过程</param>
   /// <param name="AInterval">要延迟的时间，单位为 0.1ms，如要间隔 1 秒，则值为
10000</param>
   /// <param name="AData">作业附加的用户数据指针</param>
   /// <param name="ARunInMainThread">作业要求在主线程中执行</param>
   /// <param name="AFreeType">附加数据指针释放方式</param>
   /// <returns>成功投寄返回句柄，否则返回 0</returns>
   function Delay(AProc: TQJobProc; ADelay: Int64; AData: Pointer;
     ARunInMainThread: Boolean = False;

     AFreeType: TQJobDataFreeType = jdfFreeByUser): IntPtr; overload;
```

ADelay 参数决定了作业延迟的时间长度，单位为 0.1ms。下面我们就通过 Delay 做一个随机延迟的 Demo，这上作业的内容随机间隔 1-3 秒，输出一次当前时间：

【Delphi】

```
procedure TForm1.DoRandomTimer(AJob: PQJob);

var

  ANextDelay:Int64;

begin

OutputDebugString(PChar(FormatDateTime('yyyy-mm-dd hh:nn:ss',Now())));

ANextDelay:=Q1Second+Q1Second*random(300) div 100;

Workers.Delay(DoRandomTimer,ANextDelay,nil);

end;



procedure TForm1.FormCreate(Sender: TObject);

begin

Workers.Delay(DoRandomTimer,1*Q1Second,nil);


end;
```

【C++ Builder】

```
__fastcall TForm1::TForm1(TComponent* Owner)

: TForm(Owner)

{

Workers->Delay(DoRandomTimer,1*Q1Second,NULL);

}

//---------------------------------------------------------------------
```

```
void __fastcall TForm1::DoRandomTimer(PQJob AJob)

{

OutputDebugString(FormatDateTime("yyyy-mm-dd hh:nn:ss",Now()).c_str());

__int64 ANextDelay=Q1Second+Q1Second*random(300)/100;

Workers->Delay(DoRandomTimer,ANextDelay,NULL);


}
```

3、使用 At 函数在指定的时间点执行

除了上述两种定时作业外， QWorker 还支持定点作业。所谓的定点作业就是在指定的时间点执行的某类作业，比如每天的 9:30 分执行，或者每小时的 30 分执行。

首先我们来看 At 函数的声明：

```
  /// <summary>投寄一个在指定时间才开始的重复作业</summary>

   /// <param name="AProc">要定时执行的作业过程</param>

   /// <param name="ADelay">第一次执行前先延迟时间</param>

   /// <param name="AInterval">后续作业重复间隔，如果小于等于 0，则作业只执行一次，和

Delay 的效果一致</param>

   /// <param name="ARunInMainThread">是否要求作业在主线程中执行</param>

   /// <param name="AFreeType">附加数据指针释放方式</param>

   /// <returns>成功投寄返回句柄，失败返回 0</returns>

   function At(AProc: TQJobProc; const ADelay, AInterval: Int64;

     AData: Pointer; ARunInMainThread: Boolean = False;

     AFreeType: TQJobDataFreeType = jdfFreeByUser): IntPtr; overload;
```

```
    /// <summary>投寄一个在指定时间才开始的重复作业</summary>
    /// <param name="AProc">要定时执行的作业过程</param>
    /// <param name="ATime">执行时间</param>
    /// <param name="AInterval">后续作业重复间隔，如果小于等于 0，则作业只执行一次，和
Delay 的效果一致</param>
    /// <param name="ARunInMainThread">是否要求作业在主线程中执行</param>
    /// <param name="AFreeType">附加数据指针释放方式</param>
    /// <returns>成功投寄返回句柄，失败返回 0</returns>
    function At(AProc: TQJobProc; const ATime: TDateTime;
      const AInterval: Int64; AData: Pointer; ARunInMainThread: Boolean = False;

        AFreeType: TQJobDataFreeType = jdfFreeByUser): IntPtr; overload;
```

第一种形式下，采用的和 Delay 类似的形式，效果等价于 Delay + Post 定时重复的效果。首次运行会延迟 ADelay*0.1ms，然后每次按 AInterval 的时间间隔为基准进行重复。如果 AInterval 参数指定为 0，则和 Delay 的效果是完全等价的。

第二种形式下，作业首次会在指定的时间点执行，然后再以 AInterval 指定的时间间隔进行重复作业。如果指定的时间已经过了，则计划下一次执行时间。

下面的代码将计划 DoAtJob 作业每天的 9:30 执行一次。

【Delphi】

```
Workers.At(DoAtJob,EncodeTime(9,30,0,0),Q1Day,nil);
```

【C++ Builder】

```
Workers->At(DoAtJob,EncodeTime(9,30,0,0),Q1Day,NULL);
```

下面是关于 At 函数的一些技巧提示：

1、常见的时间间隔可以直接用 Q1XXXX 乘以数量来计算，如每隔 2 小时执行一次，则 AInterval 参数为 2*Q1Hour，每隔 1 周执行一次，则 AInterval 为 7*Q1Day。

2、由于每年、每季度和每月的时长不一样，所以你无法直接用 At 函数来规划这种作业，这种作业的话，建议你直接用 At 单次作业实现，在一次作业完成时，用 IncYear 或 IncMonth 来在指定的时间点计划下一次作业。

好了，下面是关于 QWorker 中时间间隔的一个说明：

QWorker 中时间间隔的单位是 0.1ms，也就是说，在操作系统支持的情况下， QWorker 的作业最快能够 0.1ms 调度 1 次（如果你的作业执行时间本身就超过 0.1ms 的话就别考虑了），而 QWorker 使用了一个 64 位整数来对时间间隔进行计数，所以理论上，QWorker 的时间间隔最大约为 10675199116 天，约合 2923 年，所以不用担心你的时间溢出的问题。

第八章 状态机：信号与广播

前面我们讨论了定时作业（Post/Delay/At）和直接用 Post 来触发作业，也就是说，作业触发和执行之间是紧密耦合的。作业的触发者知道作业的响应者在那儿，要干什么，但如果我们要将作业的触发与执行分离，该怎么做呢？信号！QWorker 提供了信号机制来解决这一问题。

QWorker 中信号的作用就在于建立触发者和响应者之间松散耦合的中介，前面我们在作业类型的讨论中，知道了信号作业的几个步骤，我们先来回顾下：

（1）、注册信号

（2）、注册信号对应的作业

（3）、触发信号

实际上，除了第一步是必需首先执行的，步骤（2）和步骤（3）并没有顺序要求。信号的触发并不要求有信号处理作业，相应的，注册信号对应的作业与触发信号的执行当然也没多大关系。信号实际上是一种状态机，只有两种状态：

（1）、等待状态

信号一旦注册，它就已经处于等待状态。同样如果一个信号没有被触发，那么它也处于等待状态。

（2）、触发状态

信号一旦被触发，它就处于触发状态。此时，它会触发所有已注册的作业执行一遍。注意，这些被触发的作业是被并行执行的，所以作业执行的顺序与其注册的顺序无关。

一盏灯点亮了，屋里所有的人都看到了光明。信号也一样，它可以有多个作业做为信号的接收者，一个信号一旦被触发了，所有注册的作业都会被触发执行，形成广播的效果。

在设计基于信号的 QWorker 作业时，我希望提醒大家以下几点：

（1）、信息的处理作业可以运行在主线程，也可以运行在后台线程，这点和普通的作业没什么两样，只需要在使用 Wait 注册信号等待作业时，标记作业需要在主线程中执行即可。

我们来看下 Wait 函数的声明：

```
/// <summary>投寄一个等待信号才开始的作业</summary>
```

```
  /// <param name="AProc">要执行的作业过程</param>
  /// <param name="ASignalId">等待的信号编码，该编码由 RegisterSignal 函数返回</param>
  /// <param name="ARunInMainThread">作业要求在主线程中执行</param>
  /// <returns>成功投寄返回句柄，否则返回 0</returns>
  function Wait(AProc: TQJobProc; ASignalId: Integer;ARunInMainThread: Boolean = False):
IntPtr; overload;
```

ARunInMainThread 参数决定了作业在主线程和后台线程中执行。

（2）、作业触发时，参数通过信号触发函数 Signal 传递，如果指定了 AFreeType 方式，则参数会在不需要时，由 QWorker 负责释放。由于信号作业是多播并发的，具有通知的特性，所以要注意：

不要更改参数的值，也就是说，参数值只能当常量访问，否则，可能会造成与其它信号处理作业之间的冲突。

不要使用会引发参数状态变化的参数。因为它同样会影响其它作业的处理，比如 TStream 在 Read 或 Write 时都会调整当前位置，如果使用它做为参数，显示，如果你 Read 或 Write 时，其它作业处理函数如果也 Read 或 Write 处理，就会得不到正确的结果。

不要传递栈上的变量作为参数，信号作业的处理过程是异步的，不是同步的，所以不要指望你触发信号结束后，对应的作业也结束了。如果用栈上的变量，那么由于函数退出后，栈上的变量就被释放了，所以会造成意外的结果，这显然是不安全的。

综上所述，我们一定要注意信号作业的参数不能用于传出值，只能用于传入值。如果要传递返回参数，那么应该通过新的作业来传递（如触发一个新的信号，将返回值作为参数传递回去，这需要规范好作业双方的协议）。

关于信号作业的一个实例，在 QWorkerDemo 和 StateWorker 两个演示程序中，我们进行了展示。我们摘抄一段 QWorkerDemo 示例中多播的代码来进行简单的说明：

1、首先，是注册信号名称，获得一个唯一的信号 ID，这个 ID 用于唯一标记这个信号，并在整个程序的生命周期内不会变动。

【Delphi】

```
FMulticastSignal := Workers.RegisterSignal('Multicase.Start');
```

【C++ Builder】

```
FMulticastSignal = Workers->RegisterSignal(L"Multicase.Start");
```

RegisterSignal 接受的唯一一个参数是信号名称，该信号名称是整个程序唯一的，同名的重复注册会被认为是同一个信号，返回同一个 ID。我们将其值保存到 FMulticastSignal 变量中。由此，我们的作业响应函数注册的地方，就可以简单的重复调用 RegisterSignal 来获取这个 ID。

2、接下来，我们注册这个信号的作业响应函数：

【Delphi】

```
Workers.Wait(DoMulticastSingal1, FMulticastSignal);


Workers.Wait(DoMulticastSingal2, FMulticastSignal);
```

【C++ Builder】

```
Workers->Wait(DoMulticastSingal1, FMulticastSignal);


Workers->Wait(DoMulticastSingal2, FMulticastSignal);
```

这样，我们在触发 Multicase.Start 信号时，将会同时触发 DoMulticastSingal1 和 DoMulticastSingal2 两项作业。

3、最后，我们人为的来触发下这个信号。

触发信号可以按 ID 或名称来触发，我们推荐按 ID 来触发，因为按名称触发实际上触发了内部的一个额外的查找过程，效率上有所损失。我们看一下两个声明：

```
/// <summary>触发一个信号</summary>

/// <param name="AId">信号编码，由 RegisterSignal 返回</param>
```

```
    /// <param name="AData">附加给作业的用户数据指针地址</param>
    /// <param name="AFreeType">附加数据指针释放方式</param>
    /// <remarks>触发一个信号后，QWorkers 会触发所有已注册的信号关联处理过程的执行
</remarks>
    procedure Signal(AId: Integer; AData: Pointer = nil;
      AFreeType: TQJobDataFreeType = jdfFreeByUser); overload;
    /// <summary>按名称触发一个信号</summary>
    /// <param name="AName">信号名称</param>
    /// <param name="AData">附加给作业的用户数据指针地址</param>
    /// <param name="AFreeType">附加数据指针释放方式</param>
    /// <remarks>触发一个信号后，QWorkers 会触发所有已注册的信号关联处理过程的执行
</remarks>
    procedure Signal(const AName: QStringW; AData: Pointer = nil;

      AFreeType: TQJobDataFreeType = jdfFreeByUser); overload;
```

好了，我们看下触发的过程：

【Delphi】

```
procedure TForm1.Button24Click(Sender: TObject);
var
  AParams: TQMsgPack;
begin
AParams := TQMsgPack.Create;
```

```
AParams.Add('TimeStamp').AsDateTime := Now;

AParams.Add('Sender', 'Button24');

Workers.Signal(FMulticastSignal, AParams, jdfFreeAsObject);


end;
```

【C++ Builder】

```
void __fastcall TForm1::Button24Click(System::TObject *Sender);

{

TQMsgPack *AParams = new TQMsgPack;

AParams->Add(L"TimeStamp")->AsDateTime = Now();

AParams->Add(L"Sender", "Button24");

Workers->Signal(FMulticastSignal, AParams, jdfFreeAsObject);


}
```

话说，有人可能发现，信号没有取消注册函数（注意等待信号的作业本身可以直接用 Clear 系列函数取消与信号的关联）。这里这么设计有以下几个原因：

（1）、信号的注册、触发和响应分离的机制，决定了信号的反注册时机的把握对程序来说，是不好把握的。

（2）、一个信号注册信息所占用的资源并不大，而一个程序中，信号的数量是相对有限的，其由此带来的资源额外开销可以忽略不计。

当然，对于追求完美主义者来说，确实不够完美，但目前的实现就是这样，而在目前的规划中，我并不打算加入它。

第九章 作业分组

在前面的章节中，我们讨论了常见的简单作业类型的处理，这种作业如果有多步，我们就需要自己手动在作业中投递新的作业，那么我们有没有一种办法来简化这种操作呢？这就是我们接下来要讨论的内容。

作业分组在 QWorker 中使用 TQJobGroup 来管理，用于将一到多个作业给组合在一起当做一个对象来处理。这些作业有可能是串行执行，也可能是并行执行。他们的区别在于，对于并行执行的作业，相互之间没有啥顺序要求，谁先谁后无所谓，而串行执行的作业，必需挨个执行，一个执行完成后，才能轮到下一个执行（这不废话嘛:)）。

好吧，废话说了一堆，我们先来看看 TQJobGroup 的接口声明：

```
  /// <summary>构造函数</summary>
   /// <param name="AByOrder">指定是否是顺序作业，如果为 True，则作业会按依次执行
</param>
   constructor Create(AByOrder: Boolean = False); overload;
   /// <summary>析构函数</summary>
   destructor Destroy; override;
   /// <summary>取消剩下未执行的作业执行</summary>
   /// <param name="AWaitRunningDone">是否等待正在执行的作业执行完成，默认为
True</param>
   /// <remark>如果在当前作业中等待执行的作业完成，则请不要设置 AWaitRunningDone 为 True，
   /// 否则，会出现死等</remark>
   procedure Cancel(AWaitRunningDone: Boolean = True);
   /// <summary>要准备添加作业，实际增加内部计数器</summary>
   /// <remarks>Prepare 和 Run 必需匹配使用，否则可能造成作业不会被执行</remarks>
   procedure Prepare;
```

```
    /// <summary>减少内部计数器，如果计数器减为 0，则开始实际执行作业</summary>
    /// <param name="ATimeout">等待时长，单位为毫秒</param>
    procedure Run(ATimeout: Cardinal = INFINITE);
    /// <summary>添加一个作业过程，如果准备内部计数器为 0，则直接执行，否则只添加到列表
</summary>
    /// <param name="AProc">要执行的作业过程</param>
    /// <param name="AData">附加数据指针</param>
    /// <param name="AInMainThread">作业是否需要在主线程中执行</param>
    /// <param name="AFreeType">AData 指定的附加数据指针释放方式</param>
    /// <returns>成功，返回 True，失败，返回 False</returns>
    /// <remarks>添加到分组中的作业，要么执行完成，要么被取消，不运行通过句柄取消
</remarks>
    function Add(AProc: TQJobProc; AData: Pointer;
      AInMainThread: Boolean = False;
      AFreeType: TQJobDataFreeType = jdfFreeByUser): Boolean; overload;
    /// <summary>添加一个作业过程，如果准备内部计数器为 0，则直接执行，否则只添加到列表
</summary>
    /// <param name="AProc">要执行的作业过程</param>
    /// <param name="AData">附加数据指针</param>
    /// <param name="AInMainThread">作业是否需要在主线程中执行</param>
    /// <param name="AFreeType">AData 指定的附加数据指针释放方式</param>
    /// <returns>成功，返回 True，失败，返回 False</returns>
```

```
  /// <remarks>添加到分组中的作业，要么执行完成，要么被取消，不运行通过句柄取消
</remarks>
  function Add(AProc: TQJobProcG; AData: Pointer;
    AInMainThread: Boolean = False;
    AFreeType: TQJobDataFreeType = jdfFreeByUser): Boolean; overload;
{$IFDEF UNICODE}
  /// <summary>添加一个作业过程，如果准备内部计数器为 0，则直接执行，否则只添加到列表
</summary>
  /// <param name="AProc">要执行的作业过程</param>
  /// <param name="AData">附加数据指针</param>
  /// <param name="AInMainThread">作业是否需要在主线程中执行</param>
  /// <param name="AFreeType">AData 指定的附加数据指针释放方式</param>
  /// <returns>成功，返回 True，失败，返回 False</returns>
  /// <remarks>添加到分组中的作业，要么执行完成，要么被取消，不运行通过句柄取消
</remarks>
  function Add(AProc: TQJobProcA; AData: Pointer;
    AInMainThread: Boolean = False;
    AFreeType: TQJobDataFreeType = jdfFreeByUser): Boolean; overload;
{$ENDIF}
  /// <summary>等待作业完成</summary>
  /// <param name="ATimeout">最长等待时间，单位为毫秒</param>
  /// <returns>返回等待结果</returns>
```

```
    /// <remarks>WaitFor 会阻塞当前线程的执行，如果是主线程中调用，建议使用 MsgWaitFor
    /// 以保证在主线中的作业能够被执行</remarks>
    function WaitFor(ATimeout: Cardinal = INFINITE): TWaitResult;
    /// <summary>等待作业完成</summary>
    /// <param name="ATimeout">最长等待时间，单位为毫秒</param>
    /// <param name="AMsgWait">是否检查消息队列，如果不在主线程中执行，则忽略该参数
</param>
    /// <returns>返回等待结果</returns>
    /// <remarks>WaitFor 会阻塞当前线程的执行，如果是主线程中调用，建议使用 MsgWaitFor
    /// 以保证在主线中的作业能够被执行</remarks>
    function WaitFor(AMsgWait:Boolean;ATimeout: Cardinal = INFINITE):
TWaitResult;overload;
    /// <summary>等待作业完成</summary>
    /// <param name="ATimeout">最长等待时间，单位为毫秒</param>
    /// <returns>返回等待结果</returns>
    /// <remarks>如果当前在主线程中执行,MsgWaitFor 会检查是否有消息需要处理，而
    /// WaitFor 不会，如果在后台线程中执行，会直接调用 WaitFor。因此，在主线程中调用
    /// WaitFor 会影响主线程中作业的执行，而 MsgWaitFor 不会
    /// </remarks>
    function MsgWaitFor(ATimeout: Cardinal = INFINITE): TWaitResult;
    /// <summary>未完成的作业数量</summary>
    property Count: Integer read GetCount;
```

```
/// <summary>全部作业执行完成时触发的回调事件</summary>

property AfterDone: TNotifyEvent read FAfterDone write FAfterDone;

/// <summary>是否是按顺序执行，只能在构造函数中指定，此处只读</summary>

property ByOrder: Boolean read FByOrder;

/// <summary>用户自定的分组附加标签</summary>

property Tag: Pointer read FTag write FTag;

/// <summary>是否在作业完成后自动释放自身</summary>


property FreeAfterDone: Boolean read FFreeAfterDone write FFreeAfterDone;
```

1、构造函数 Create

TQJobGroup 的构造函数接受一个唯一的参数 AByOrder 来决定分组内的作业是否需要按顺序执行。如果该值为 True ，则后面 Add 时添加的子作业将都是顺序执行的。

2、析构函数 Destroy

TQJobGroup 的析构函数这个唯一要说的是，它在析构时会取消所有未执行的作业，并等待正在执行的作业完成，所以，嘿嘿，你懂的，不要在 TQJobGroup 的子作业里释放分组对象，否则后果就是死循环。

3、取消未执行的分组作业函数 Cancel

Cancel 取消所有后续未进入执行状态的作业，如果作业已经执行，则是否等待执行完成，取决于 AWaitRunningDone 参数，默认是等待。但如果你在分组作业的子作业中取消，那么你就必需传入 False，以避免死循环。

4、准备批量添加作业函数 Prepare

Prepare 将使 TQJobGroup 对象进入批量添加作业状态，它会增加内部的计数器，以避免此后调用 Add 函数加的作业在调用 Run 函数之前执行。这样可以减少资源冲突，加快添加速度。在这里要注意一点，Prepare 和 Run 必需配对使用，否则，你的作业可能永远得不到执行机会。

5、开始运行分组内作业函数 Run

Run 将减少由于 Prepare 增加的引用计数，如果这个计数器减为 0，则会将作业提交给 Workers 这个包工头去分配给实际的工人处理。注意其中有一个超时设置，这个超时值的单位为毫秒，面向的是整个作业分组，也就是说，所有的作业总的执行时间不能超过这个时间。一旦超时，后续未执行的作业将会被取消，得不到执行的机会，并触发 AfterDone 事件。如果调用 WaitFor 等待中，则会触发结束等待并返回 wrTimeout。

6、添加作业处理函数 Add

Add 为分组添加一个子作业，这个作业只能是立即执行类型的作业，不能是定时或基于信号的作业。作业的具体参数可以参考普通作业 Post 函数的声明，它们是统一的。

如果未进入批量添加状态，则新增的作业根据构造函数的 AByOrder 参数，可能会立即执行（ AByOrder=False 或是第一个作业），也可能会被计划到后面执行。

7、等待作业处理完成函数 WaitFor/MsgWaitFor

这两个函数的作业都是等待作业完成直到超时，默认超时值单位为毫秒。注意它与 Run 里那个超时的区别是一致的，任意一个超时都会造成分组超时并取消后续作业的执行。

8、未执行的作业数量 Count

用于统计分组中尚未交付给 Workers 执行的作业数量。

9、分组所有作业执行完成事件 AfterDone

在所有作业执行完成时会触发该事件，如果指定了，则在所有作业执行完成或者被清队干净时会被触发。

10、是否按顺序执行 ByOrder

这个属性是只读的，值由构造函数传入。

11、用户附加标签 Tag

这个 Tag 的值及其管理由用户自行负责，它可以用于传递一些额外的信息给分组中的作业。

12、是否在所有作业完成后自动释放自身 FreeAfterDone

这一属性一般和 Run 结合在一起使用，以便在所有作业结束时，自动释放对象自身。

好了，上面说了一堆，下面说一下分组作业的一般调用流程：

1、创建 TQJobGroup 对象，并确定它的类型（是顺序还是并行），假设我们初始化 AByOrder 参数为 True，按顺序执行一组作业，则创建的对应代码如下：

【Delphi】

```
var

 AGroup:TQJobGroup;

begin

AGroup:=TQJobGroup.Create(true)


...
```

【C++ Builder】

```
TQJobGroup *AGroup=new TQJobGroup(true);
```

当然，接下来就可以设置其它属性，如设置 AfterDone、FreeAfterDone 等属性的值。

2、调用 Prepare 和 Add 添加作业：

【Delphi】

```
AGroup.Prepare;

AGroup.Add(DoStep1,nil);

AGroup.Add(DoStep2,nil);

...


AGroup.Add(DoStepn,nil);
```

【C++ Builder】

```
AGroup->Prepare();

AGroup->Add(DoStep1,NULL);

AGroup->Add(DoStep2,NULL);

...


AGroup->Add(DoStepn,NULL);
```

3、调用 Run 来启动作业，一般情况下不需要提供 Run 里的超时参数（默认永远等待）。

【Delphi】

```
AGroup.Run;
```

【C++ Builder】

```
AGroup->Run()
```

4、如果是设置了 AfterDone 事件并准备异步等待所有作业处理完，则从步骤开始可以省略。如果不是，那么我们就需要调用 WaitFor 或 MsgWaitFor 来等待分组作业执行完成。

选择使用 WaitFor 或 MsgWaitFor 取决于下面两步判断的结果：

（1）、当前代码是否运行在主线程，如果不是，WaitFor 和 MsgWaitFor 没有任何区别，MsgWaitFor 会直接调用 WaitFor 死等，此时两者皆可。

（2）、当前程序是否有运行在主线程中的作业，如果有，那么使用 MsgWaitFor 以避免阻塞主线程作业的处理，如果没有，可以选择 WaitFor，但显然由于 WaitFor 也会阻塞其它主线程消息的处理，所以会造成 UI 在等待结束前假死的现象，请斟酌选择。

【Delphi】

```
AGroup.WaitFor;
```

【C++ Builder】

```
AGroup->WaitFor();
```

5、如果没有指定 FreeAfterDone 为 True，则在需要时释放 TQJobGroup 对象实例，以避免内存泄露。

下面是两个常问的问题：

1、如果没有调用 Prepare 直接调用 Add 添加会有什么后果？

没啥后果，只是相当于添加后立即就调度执行，如果添加多个作业，有可能产生多次触发 AfterDone 事件的问题。当然，如果你再同时设置了 FreeAfterDone，那就可能会出问题了，AV 错误在等着你（有可能在你添加后一条记录之前，触发了释放代码）。

2、Run 的超时和 WaitFor 的超时有啥区别？

认真看前面 Run 函数的说明。

第十章 使用分组作为业务处理队列

作业分组在 QWorker 中是一个重要的组成部分，它是对作业的一个二次封装，以方便上层应用进行调用。但也因为它的封装的特性，相对于直接管理可能会引入一些额外的开销，但与易用性相比，许多时候这点开销我们都忽略不计了。

在本节内容中，我们继续上一个话题，这个话题是由于根据群里广大群友实际遇到的问题，做的一个简单示例。

本示例利用 TQJobGroup 可以顺序执行的特性，将每个作业分组当成一个特定类型的作业池，当有新作业需要处理时，直接将作业加入到这个分组。

首先，我们声明 TQJobGroup 一个实例 FMyPool，作为一个处理池序列，这个池中的作业会挨个被执行。

【Delphi】

```
TForm1=class(TForm)

...

private

  FMyPool:TQJobGroup;

...

end;

...

procedure TForm1.FormCreate(Sender:TObject);

begin

FMyPool:=TQJobGroup.Create(True);

...


end;
```

【C++ Builder】

```
class PACKAGE TForm1:public TForm

{

...

private

   TQJobGroup *FMyPool;

...

};

...

__fastcakk TForm1::TForm1(Owner:TComponent):TForm(Owner)

{

FMyPool:=new TQJobGroup(true);

...


}
```

接下来的事情实际上就简单了，我们只需要在必要时，将作业直接添加到 FMyPool 里就可以了。

【Delphi】

```
procedure TForm1.DoMyJob(AJob:PQJob);

begin

OutputDebugString('My Job done');

end;
```

```
procedure TForm1.Button1Click(Sender:TObject)

begin

FMyPool.Add(DoMyJob,nil);


end;
```

【C++ Builder】

```
void __fastcall TForm1::DoMyJob(PQJob AJob)

{

OutputDebugStringW(L"My Job done");

}



void __fastcall TForm1::Button1Click(System::TObject *Sender)

{

FMyPool->Add(DoMyJob,NULL);


}
```

好了，如果我们需要有多个队列，那么多建几个 TQJobGroup 对象当作作业的队列处理器就好了。当然了，最后别忘了要做好犯罪现场的清理工作，防止内存泄漏：

【Delphi】

```
procedure TForm1.FormDestroy(Sender:TObject);

begin

FreeAndNil(FMyPool);
```

```
end;
```

【C++ Builder】

```
void __fastcall TForm1::FormDestroy(System::TObject *Sender)

{

delete FMyPool;


}
```

第十一章 工作流控制

工作流（WorkFlow）是一个被炒了很久的东西，并出现了一些工作流引擎。那么，QWorker 是不是也能做一些这方面的工作呢？答案是显然的，QWorker 本身基于作业的特性，决定了它很适合完成一些普通的工作流管理任务。

工作流引擎实际上实现了两个部分的工作：流程的控制和作业的执行。工作流程的控制实际上可以分成顺序执行、条件跳转（含重复执行）两大类，而工作的执行则和我们普通的作业没有任何区别。所以，当我们用 QWorker 做为一个工作流管理的基本引擎时，我们需要做的一件很重要的事情就是如何来完成工作流控制部分。

一个最简单的工作流控制的实例是顺序执行的多步作业，这个我们直接通过创建 TQJobGroup 的顺序执行的作业分组实例即可达到目的。

稍微复杂一点的工作流控制实例是然后是顺序执行的多步作业，但是某些步骤我们会由于某些原因跳过执行。这种情况下，我们只需要在上面的基础上，在作业执行代码中，前置一些简单的条件判断处理，决定实际的作业执行代码是否执行即可达到目的。当然，考虑到流程控制的后期可定制性，显然使用类似于 PaxScript、FastScript 等脚本引擎与作业结合，能够更好的完成后期定制工作。

最复杂的工作流程控制实际上牵涉到循环的重复多步作业。比如：我们提交一个申请给上级主管部门，主管部门给出反馈意见，让我们修改申请，我们修改后再次提交上级主管部门审核，重复上述流程，直到主管部门通过或者直接驳回申请。这样一个重复的作业过程，在 QWorker 中，我们实际上可以通过信号来完成这类复杂的作业的控制的。以上面我们提到的样本为例子，我们如果以 QWorker 为流程的基本引擎，那么我们定义几个信号：

1、Request.Post – 当我们向上级提交完申请后触发；

2、Request.EditNeeded – 当上级给出修改意见后触发；

3、Request.Accept – 当上级通过我们的申请后触发；

4、Request.Deny – 当上级驳回我们的申请后触发；

5、Request.Discard – 当我们决定放弃申请时触发；

那么，现在事情的处理就相当简单了：

1、我们有个一申请编辑的作业，它用来编辑申请内容。当我们第一次提交申请时，或者收到 Request.EdtNeeded 信号时触发。在编辑完成后，我们有两个选择，一个是我们觉得我们的申请有问题或者不再需要申请了，那么我们就触发一个 Request.Discard 信号，触发放弃申请的作业，另一个是我们确认申请内容

没有问题了，触发一个 Request.Post 事件，将申请提交给上级。

2、我们的上级有一个审核作业，它在收到下级的 Request.Post 信号时触发，根据上级做出的审核结果，分别触发下级的 Request.EditNeeded、Request.Accept 或 Request.Deny 三个信号之一。

3、下级分别有两个作业对应于 Request.Accept 和 Request.Deny 两个信号，用以在上级通过和拒绝申请时，提供必要的处理。

这样子，我们就形成了一个事实上的很复杂的业务流程处理逻辑。实际生活中，再复杂的业务逻辑也可以通过上述方式进行处理。

我们再考虑工作流控制中的分支和归并问题。一个作业可能分割出多个子作业，然后在各个子作业完成后，归并到同一个部门去汇总归档。这个问题的处理实际上通过信号我们一样可以完成，作业分隔我想不用赘述，直接分隔成多个作业就好了，作业合并多说一句，简单的就在每个子作业完成后，触发一个归并请求信号如叫 Jobs.MergeNeeded ，然后在这个信号的响应作业中去检查是否所有子作业都执行完成了，如果完成了，就触发一个实际的合并信号，如 Jobs.DoMerge 来执行实际的合并作业。

实际上实现一个完整的工作流控制，需要编写的代码并不少，上面讨论中忽略了信号在网络中传递的过程，实际上，它是必不可少的环节。但我们讨论的主要是利用 QWorker 作业一个流程控制的基础引擎的实现方式，所以，我们省略了具体的实现，而只是讨论了理论上应该如何实现。一个完整的工作流引擎还需要做许多工作，不是本文讨论的范围。当然，你可以用 QWorker + PaxScript + DIOCP 实现一个作业调度、脚本化和网络传送，从而实现一个真正完整的工作流引擎。

第十二章 For 并行

看到这个标题，你可能会感觉到疑惑：难道我们前面用 Post 或 TQJobGroup 提交的作业不是并行的吗？为什么又会冒出一个 For 并行？

首先，第一个问题的答案是：它们当然是并行的，但在特定的场合下，由于调度会引起一定的性能损失。

接下来，第二个问题的答案是：For 并行是在特定场合下的解决方案，它的目的是按指定的次数重复执行指定的作业处理函数。由于它不需要重复投寄作业，所以调度开销更小，在满足条件的前提下，性能更佳。

我们来看看 For 并行有什么限制：

（1）、它只会每个 CPU 使用一个工作者工作。

（2）、它必需等待所有的作业完成或调用 BreakIt 中止时，才会退出执行。

（3）、它的所有作业执行函数都是在后台线程中的执行的，不允许调度主线程作业。

（4）、同时只有一个作业函数在并发执行，不能同时运行不同的作业函数。

而与 For 并行相比，TQJobGroup 的主要优势体现在：

（1）、更大的并发处理能力，实际的工作者数量仅受作业数量和 MaxWorkers 属性限制。

（2）、更灵活。

（2.1）、作业可以工作在主线程和后台线程中；

（2.2）、作业可以顺序执行，也可以并行执行；

（2.3）、可以同时运行不同的作业函数。

综上所述，在能够使用 For 并行的场合，性能要比 TQJobGroup 更高。我们根据上面的比较可以基本确定一些原则：

（1）、For 不能用于需要在主线程中执行的作业；

（2）、For 适用于同时执行同一个函数的作业，同时运行不同的作业处理函数需要使用 TQJobGroup；

（3）、For 不能要求作业函数的执行顺序；

（4）、For 不应用于慢速 IO 牵涉到的计算；

好了，说了这么多，我们来做一个实际的 For 并行的例子，首先来看 For 并行的声明：

```
class function TQWorkers.&For(const AStartIndex, AStopIndex: NativeInt;
  AWorkerProc: TQForJobProc; AMsgWait: Boolean; AData: Pointer;
  AFreeType: TQJobDataFreeType): TWaitResult;inline;
class function TQWorkers.&For(const AStartIndex, AStopIndex: NativeInt;
  AWorkerProc: TQForJobProcA; AMsgWait: Boolean; AData: Pointer;
  AFreeType: TQJobDataFreeType): TWaitResult;inline;
class function TQWorkers.&For(const AStartIndex, AStopIndex: NativeInt;
  AWorkerProc: TQForJobProcG; AMsgWait: Boolean; AData: Pointer;AFreeType:
TQJobDataFreeType): TWaitResult;inline
```

你可能注意到了，这三个 For 函数都是 inline 内联函数，它实际上触发的是 TQForJobs 的方法，这只是统一接口的语法糖，两者本身没有任何区别。我们来了解下各个参数的含义：

AStartIndex：循环起始值

AStopIndex：循环结束值

AWorkerProc：作业处理过程，有三种重载，分别对应类成员函数、匿名函数和全局函数版本。

AMsgWait：是否在等待时允许消息循环，如果为 True，则主线程不会被阻塞，消息能够正常处理，程序自己处理必要的阻塞，否则，主线程被阻塞。

AData：作业附加数据，被当做作业的 Data 成员传递给作业处理函数。

AFreeType：作业附加数据类型，用于决定是否自动释放作业的附加数据，默认由用户负责释放。

在执行完成后，会返回一个等待结果，如果成功从 AStartIndex 循环到 AStopIndex，则返回 wrSignaled，如果被中途取消（调用 Break，或者程序退出），则返回 wrAbandoned。

【注意】实际循环的执行范围是集合 [AStartIndex,AStopIndex]，这意味着 AStopIndex 应该大于等于 AStartIndex，否则循环过程将不会被执行。

那么，接下来，我们看下 For 作业函数的声明：

```
TQForJobProc = procedure(ALoopMgr: TQForJobs; AJob: PQJob; AIndex: NativeInt) of object;

TQForJobProcG = procedure(ALoopMgr: TQForJobs; AJob: PQJob;AIndex: NativeInt);


TQForJobProcA = reference to procedure(ALoopMgr: TQForJobs; AJob: PQJob;AIndex:

NativeInt);
```

与普通的作业过程相比，For 循环作业过程多了两个参数：

（1）、ALoopMgr 参数是循环作业的控制对象，它提供了一个 BreakIt 方法用于中断循环的执行；

（2）、AIndex 参数指明了当前循环的索引值，表明它是第几个循环。

至于 AJob 参数，其含义与普通的作业没有任何区别。

下在我们来看具体的例子，同前面一样，我们省略了匿名函数和全局函数的版本，需要的话，您 只需要将相关函数直接替换即可：

【Delphi】

```
procedure TForm1.DoForJobProc(AMgr: TQForJobs; AJob: PQJob; AIndex: NativeInt);

begin

InterlockedIncrement(PInteger(AJob.Data)^);

end;



procedure TForm1.Button1Click(Sender:TObject);
```

```
var

  ARuns:NativeInt;

  T,T1:Cardinal;

  I: Integer;

begin

ARuns:=0;

T:=GetTickCount;

TQForJobs.For(0,999999,DoForJobProc,False,@ARuns);

T:=GetTickCount-T;

ShowMessage(IntToStr(T)+'ms');


end;
```

【C++ Builder】

```
void __fastcall TForm1::DoForJobProc(TQForJobs *AMgr,PQJob *AJob,NativeInt AIndex)

{

InterlockedIncrement(*((int *)AJob.Data));

}



void __fastcall TForm1::Button1Click(System::TObject *Sender);

{

NativeInt ARuns;

DWORD T,T1;
```

```
Integer I;

ARuns=0;

T=GetTickCount();

TQWorkers::For(0,999999,DoForJobProc,false,&ARuns);

T=GetTickCount()-T;

ShowMessage(IntToStr(T)+'ms');


}
```

关于 For 并行，本文暂时只说这么多，如果有什么疑问，欢迎在官方群或者后面留言。

第十三章 计划任务

如果前面的各种设置无法满足你的要求，你可能需要的是计划任务。QWorker 支持 Linux 的 Cron 计划任务配置文件格式，可以直接调用 Plan 函数来传递格式字符串到程序中进行任务调度。

与普通的作业不同，计划的作业任务分辨率比较粗，与 Linux Cron 配置文件格式一样精确到分钟。

QWorker 支持的 Linux 的 Cron 格式说明：

分 时 日 月 周 命令

如果某一位置值为“*”，则代表忽略这一条件，只要其它条件符合就执行;

支持 n-n 格式的范围，如 3-6 代表从 3 到 6;

支持间隔设置，格式为/n，如/2;

周后面的命令不支持换行，后面所有内容直接被当做命令保存下来，赋给 TQPlanMask.Content 成员。

下面是几个例子：

1. 每分钟执行一次： * * * * *
2. 每小时 5 分执行一次：5 * * * *
3. 每个月 1 号 23:00 执行一次：0 23 1 * *
4. 每周一晚上 23:00 执行一次：0 23 * * 1
5. 每小时的 30-40，每隔两分钟执行一次：30-40/2 * * * *

QDAC 对命令格式的检查进行了一些较宽容的处理，如下面的格式也认为是合适的值：

3 这是计划任务

上面创建的是每小时的 3 分执行的计划任务。也就是说，遇到无效的字符就认为是命令的开始。

现在看下 QWorker 的 Plan 函数声明：

【Delphi】

```
function Plan(AProc: TQJobProc; const APlan: TQPlanMask; AData:
Pointer;ARunInMainThread: Boolean = False;AFreeType: TQJobDataFreeType =
```

```
jdfFreeByUser): IntPtr; overload;

function Plan(AProc: TQJobProc; const APlan: QStringW; AData:
Pointer;ARunInMainThread: Boolean = False;AFreeType: TQJobDataFreeType =
jdfFreeByUser): IntPtr; overload;

function Plan(AProc: TQJobProcG; const APlan: TQPlanMask; AData:
Pointer;ARunInMainThread: Boolean = False;AFreeType: TQJobDataFreeType =
jdfFreeByUser): IntPtr; overload;

function Plan(AProc: TQJobProcG; const APlan: QStringW; AData:
Pointer;ARunInMainThread: Boolean = False;AFreeType: TQJobDataFreeType =
jdfFreeByUser): IntPtr; overload;

function Plan(AProc: TQJobProcA; const APlan: TQPlanMask; AData:
Pointer;ARunInMainThread: Boolean = False;AFreeType: TQJobDataFreeType =
jdfFreeByUser): IntPtr; overload;

function Plan(AProc: TQJobProcA; const APlan: QStringW; AData:
Pointer;ARunInMainThread: Boolean = False;AFreeType: TQJobDataFreeType =
jdfFreeByUser): IntPtr; overload;
```

【C++】

```
NativeInt __fastcall Plan(TQJobProc AProc, const Qtimetypes::TQPlanMask &APlan, void *
AData, bool ARunInMainThread = false, TQJobDataFreeType AFreeType =
(TQJobDataFreeType)(0x0))/* overload */;

NativeInt __fastcall Plan(TQJobProc AProc, const System::UnicodeString APlan, void *
AData, bool ARunInMainThread = false, TQJobDataFreeType AFreeType =
(TQJobDataFreeType)(0x0))/* overload */;

NativeInt __fastcall Plan(TQJobProcG AProc, const Qtimetypes::TQPlanMask &APlan, void *
AData, bool ARunInMainThread = false, TQJobDataFreeType AFreeType =
(TQJobDataFreeType)(0x0))/* overload */;

NativeInt __fastcall Plan(TQJobProcG AProc, const System::UnicodeString APlan, void *
AData, bool ARunInMainThread = false, TQJobDataFreeType AFreeType =
(TQJobDataFreeType)(0x0))/* overload */;

NativeInt __fastcall Plan(_di_TQJobProcA AProc, const Qtimetypes::TQPlanMask &APlan,
void * AData, bool ARunInMainThread = false, TQJobDataFreeType AFreeType =
```

```
TQJobDataFreeType)(0x0))/* overload */;

NativeInt __fastcall Plan(_di_TQJobProcA AProc, const System::UnicodeString APlan, void *
AData, bool ARunInMainThread = false, TQJobDataFreeType AFreeType =
(TQJobDataFreeType)(0x0))/* overload */;
```

其中的 APlan 参数可以是 TQPlanMask 或者是 Linux Cron 兼容的计划任务字符串，内部自动解析处理。至于其它参数和普通的 Post 作业完全一样，也不就再解释。

【Delphi】

```
Workers.Plan(DoPlanJob,TQPlanMask.Create('* * * * * 1 Minute Plan Job'),nil,true);
```

和

```
Workers.Plan(DoPlanJob,'* * * * * 1 Minute Plan Job',nil,true);
```

效果等价，其中的 DoPlanJob 函数：

```
procedure TForm1.DoPlanJob(AJob: PQJob);

var

  APlan:PQJob;

begin

APlan:=AJob.PlanJob;

lblPlanStatic.Caption:='计划任务已执行'+IntToStr(AJob.Runs+1)+'次'#13#10+'内

容:'+APlan.ExtData.AsPlan.Plan.Content;

end;
```

【C++】

```
Workers->Plan(DoPlanJob,TQPlanMask.Create(L"* * * * * 1 Minute Plan Job"),NULL,true);
```

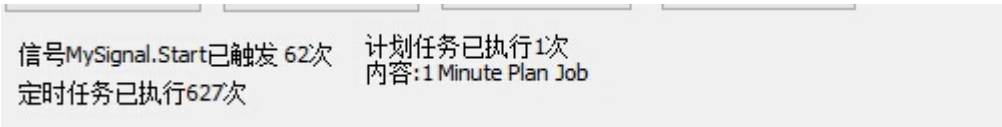
和

```
Workers->Plan(DoPlanJob,"* * * * * 1 Minute Plan Job",NULL,true);
```

效果等价，其中的 DoPlanJob 函数：

```
void __fastcall TForm1::DoPlanJob(PQJob AJob);

    {

    PQJob APlan=PQJob(AJob.PlanJob);

    lblPlanStatic->Caption=L"计划任务已执行"+IntToStr(AJob.Runs+1)+L"次\r\n 内容:'+APlan->ExtData->AsPlan->Plan->Content;

    }
```

实际执行效果：



第十四章 状态跟踪

当我们将一个作业的执行委托给 Workers 这个包工头以后，那么这个作业实际上就处于三种状态之一：

排队中：作业在队列中等待被调度执行

运行中：作业正在执行

已完成：作业已经执行完，此时作业及相关连的数据都已自动被释放

由于已经完成的作业会被释放掉，所以，实际上，我们能得到的状态只有排队中和运行中两种，第三种由于作业已经不存在，我们就无法检测到其状态。

QWorker 提供了两个函数来跟踪作业的状态：PeekJobState 和 EnumJobStates，前者针对的是单个具体的作业，后者针对的是全部作业。在具体了解这两个函数之前，我们先看一下我们能得到的作业状态信息定义：

【Delphi】

```
/// <summary>作业状态</summary>

 TQJobState = record

   Handle: IntPtr; // 作业对象句柄

   Proc: TQJobMethod; // 作业过程

   Flags: Integer; // 标志位

   IsRunning: Boolean; // 是否在运行中，如果为 False，则作业处于队列中

   Runs: Integer; // 已经运行的次数

   EscapedTime: Int64; // 已经执行时间

   PushTime: Int64; // 入队时间

   PopTime: Int64; // 出队时间

   AvgTime: Int64; // 平均时间

   TotalTime: Int64; // 总执行时间
```

```
    MaxTime: Int64; // 最大执行时间

    MinTime: Int64; // 最小执行时间

    NextTime: Int64; // 重复作业的下次执行时间


  end;
```

【C++ Builder】

```
struct DECLSPEC_DRECORD TQJobState

{

public:

NativeInt Handle;

TQJobMethod Proc;

int Flags;

bool IsRunning;

int Runs;

__int64 EscapedTime;

__int64 PushTime;

__int64 PopTime;

__int64 AvgTime;

__int64 TotalTime;

__int64 MaxTime;

__int64 MinTime;

__int64 NextTime;
```

```
};
```

在调用 PeekJobState 时，如果找到作业，则 Handle 与传入的作业句柄一致，Proc 指向作业对应的过程句柄，Flags 是作业的标志位，保存的是 JOB_XXX 标志位，可以通过位与的方式来检查各个标志位是否设置，具体标志位定义参考下表：

JOB_RUN_ONCE	作业只运行一次
JOB_IN_MAINTHREAD	作业只能在主线程中运行
JOB_MAX_WORKERS	尽可能多的开启可能的工作者线程来处理作业，暂不支持
JOB_LONGTIME	作业需要很长的时间才能完成，以便调度程序减少它对其它作业的影响
JOB_SIGNAL_WAKEUP	作业根据信号需要唤醒
JOB_TERMINATED	作业不需要继续进行，可以结束了
JOB_GROUPED	当前作业是作业组的一员
JOB_ANONPROC	当前作业过程是匿名函数
JOB_FREE_OBJECT	Data 关联的是 Object，作业完成或清理时释放
JOB_FREE_RECORD	Data 关联的是 Record，作业完成或清理时释放
JOB_FREE_INTERFACE	Data 关联的是 Interface，作业完成时调用_Release
JOB_FREE_CUSTOM1	Data 关联的成员由用户指定的方式 1 释放
JOB_FREE_CUSTOM2	Data 关联的成员由用户指定的方式 2 释放
JOB_FREE_CUSTOM3	Data 关联的成员由用户指定的方式 3 释放
JOB_FREE_CUSTOM4	Data 关联的成员由用户指定的方式 4 释放
JOB_FREE_CUSTOM5	Data 关联的成员由用户指定的方式 5 释放
JOB_FREE_CUSTOM6	Data 关联的成员由用户指定的方式 6 释放
JOB_DATA_OWNER	作业是 Data 成员的所有者

好了，现在我们来看 PeekJobState 的定义：

【Delphi】

```
/// <summary>获取指定作业的状态</summary>

/// <param name="AHandle">作业对象句柄</param>

/// <param name="AResult">作业对象状态</param>

/// <returns>如果指定的作业存在，则返回 True，否则，返回 False</returns>

/// <remarks>
```

```
  /// 1.对于只执行一次的作业，在执行完后不复存在，所以也会返回 false
  /// 2.在 FMX 平台，如果使用了匿名函数作业过程，必需调用 ClearJobState 函数来执行清理过程，
以避免内存泄露。
  /// </remarks>

  function PeekJobState(AHandle: IntPtr; var AResult: TQJobState): Boolean;
```

【C++ Builder】

```
bool __fastcall PeekJobState(NativeInt AHandle, TQJobState &AResult);
```

请注意其中的注释中的 Remark 部分说明，PeekJobState 的返回值为安全起见，建议使用 ClearJobState 来释放返回的结果，以使匿名函数的引用计数在移动平台工作正常。

我们看下 ClearJobState 的实现：

```
procedure ClearJobState(var AState: TQJobState);
begin
if IsFMXApp then
 begin
 if (AState.Flags and JOB_ANONPROC) <> 0 then
  begin
  IUnknown(AState.Proc.ProcA)._Release;
  end;
 AState.Proc.Code := nil;
 AState.Proc.Data := nil;
 end;
```

```
end;
```

可以看到，只是在 FMX 平台它起作业，VCL 中，由于 ProcA 定义为 TQJobProcA 所以会自动释放（FMX 平台 ProcA 的定义为 Pointer，定义为 TQJobProcA 无法编译）。

下面的代码是 PeekJobState 的一个例子：

```
var
  AState: TQJobState;
  S: String;
  ATime: Int64;
  ALoc: TQSymbolLocation;
begin
ATime := GetTimeStamp;
if Workers.PeekJobState(FSignalWaitHandle, AState) then
  begin
  S := '作业 ';
  if (AState.Flags and JOB_ANONPROC) = 0 then
    begin
    if LocateSymbol(AState.Proc.Code, ALoc) then
      S := S + ' - ' + ALoc.FunctionName
    else
      S := S + TObject(AState.Proc.Data).MethodName(AState.Proc.Code);
    end
```

```
else
  S := S + ' - 匿名函数';
if AState.IsRunning then
  S := S + ' 运行中:'#13#10
else
  S := S + ' 计划中:'#13#10;
case AState.Handle and $03 of
  0:
    begin
    S := S + ' 简单作业'#13#10;
    ShowMessage(S);
    Exit;
    end;
  1:
    S := S + ' 重复作业(距下次执行时间: ' + FormatFloat('0.#',
      (AState.NextTime - ATime) / 10) + 'ms)'#13#10;
  2:
    S := S + ' 信号作业'#13#10;
end;
S := S + ' 已运行:' + IntToStr(AState.Runs) + ' 次'#13#10 + ' 任务提交时间:' +
  RollupTime((ATime - AState.PushTime) div 10000) + ' 前'#13#10;
if AState.PopTime <> 0 then
```

```
    S := S + ' 末次执行时间:' + RollupTime((ATime - AState.PopTime) div 10000) +
      ' 前' + #13#10;
  S := S + ' 平均每次用时:' + FormatFloat('0.#', AState.AvgTime / 10) + 'ms'#13#10
    + ' 总计用时:' + FormatFloat('0.#', AState.TotalTime / 10) + 'ms'#13#10 +
    ' 最大用时:' + FormatFloat('0.#', AState.MaxTime / 10) + 'ms'#13#10 +
    ' 最小用时:' + FormatFloat('0.#', AState.MinTime / 10) + 'ms'#13#10;
  S := S + ' 标志位:';
  if (AState.Flags and JOB_RUN_ONCE) <> 0 then
    S := S + '单次,';
  if (AState.Flags and JOB_IN_MAINTHREAD) <> 0 then
    S := S + '主线程,';
  if (AState.Flags and JOB_GROUPED) <> 0 then
    S := S + '已分组,';
  if S[Length(S)] = ',' then
    SetLength(S, Length(S) - 1);
  ShowMessage(S);
  ClearJobState(AState);
  end
else
  ShowMessage('未找到请求的句柄对应的作业，作业可能已经完成。');

end;
```

当然，上面的例子由于用到了 QMapSymbols 单元的函数，所以只能在 Delphi 的 Windows 下程序中运行。

与 PeekJobState 不同，EnumJobStates 是返回一个动态数组来包含所有的作业的状态，声明如下：

【Delphi】

```
  /// <summary>枚举所有的作业状态</summary>

   /// <returns>返回作业状态列表</summary>

   /// <remarks>在 FMX 平台，如果使用了匿名函数作业过程，必需调用 ClearJobStates 函数来执行清理过程</remarks>

    function EnumJobStates: TQJobStateArray;
```

【C++ Builder】

```
TQJobStateArray __fastcall EnumJobStates(void);
```

这个和 PeekJobState 很类似，就不再赘述，直接上示例代码：

```
var

 AStates: TQJobStateArray;

 I: Integer;

 ATime: Int64;

 ALoc: TQSymbolLocation;

 ABuilder:TQStringCatHelperW;

begin

ATime := GetTimeStamp;

AStates := Workers.EnumJobStates;
```

```
ABuilder:=TQStringCatHelperW.Create;

ABuilder.Cat('共发现 ').Cat(Length(AStates)).Cat(' 项作业'#13#10);

for I := 0 to High(AStates) do

 begin

 if ABuilder.Position>1000 then

  begin

  ABuilder.Cat('...(后续省略)');

  Break;

  end;

 ABuilder.Cat('作业 #').Cat(I + 1);

 if (AStates[I].Flags and JOB_ANONPROC) = 0 then

  begin

  if LocateSymbol(AStates[I].Proc.Code, ALoc) then

   ABuilder.Cat(' - ').cat(ALoc.FunctionName)

  else

   ABuilder.Cat(TObject(AStates[I].Proc.Data).MethodName(AStates[I].Proc.Code));

  end

 else

  ABuilder.Cat(' - 匿名函数');

 if AStates[I].IsRunning then

  ABuilder.Cat(' 运行中:'#13#10)

 else
```

```
  ABuilder.Cat(' 计划中:'#13#10);
 case AStates[I].Handle and $03 of
  0:
   begin
   ABuilder.Cat(' 简单作业'#13#10);
   Continue;
   end;
  1:
   ABuilder.Cat(' 重复作业(距下次执行时间: ' + FormatFloat('0.#',
     (AStates[I].NextTime - ATime) / 10) + 'ms)'#13#10);
  2:
   ABuilder.Cat(' 信号作业'#13#10);
 end;
 ABuilder.Cat(' 已运行:').Cat(AStates[I].Runs).Cat(' 次'#13#10).Cat(' 任务提交时间: ').Cat(
  RollupTime((ATime - AStates[I].PushTime) div 10000)).Cat(' 前'#13#10);
 if AStates[I].PopTime <> 0 then
  ABuilder.Cat(' 末次执行时间: ').Cat(RollupTime((ATime - AStates[I].PopTime) div
10000)).Cat(
   ' 前'#13#10);
 ABuilder.Cat(' 平均每次用时:').Cat(FormatFloat('0.#', AStates[I].AvgTime /
10)).Cat('ms'#13#10)
  .Cat(' 总计用时:').Cat(FormatFloat('0.#', AStates[I].TotalTime / 10)).Cat('ms'#13#10).Cat(
```

```
    ' 最大用时:').Cat(FormatFloat('0.#', AStates[I].MaxTime / 10)).Cat('ms'#13#10).Cat(
    ' 最小用时:').Cat(FormatFloat('0.#', AStates[I].MinTime / 10)).Cat('ms'#13#10);
  ABuilder.Cat(' 标志位:');
  if (AStates[I].Flags and JOB_RUN_ONCE) <> 0 then
    ABuilder.Cat('单次,');
  if (AStates[I].Flags and JOB_IN_MAINTHREAD) <> 0 then
    ABuilder.Cat('主线程,');
  if (AStates[I].Flags and JOB_GROUPED) <> 0 then
    ABuilder.Cat('已分组,');
  ABuilder.Cat(SLineBreak);
  end;
ClearJobStates(AStates);
ShowMessage(ABuilder.Value);
FreeObject(ABuilder);

end;
```

在这个示例中，限制了下最多显示前 1000 个字节的内容，否则太长了，用 ShowMessage 显示会造成假死的现象。

第十五章 同步与锁定

凡是多线程编程，几乎就离不开同步和锁定这个话题。在深入探讨之前，首先我们了解下同步和锁定是怎么回事：

同步是为了在多线程中串行化对公共资源的访问而采取的一种策略，相关的对象在 Delphi 中在 syncobjs 和 sysutils 单元。一般来说，多线程读取在整个应用程序生存周期内，始终保持不变的公共资源是安全，就象一个广播，一群人都可以听到相关的信息，而不用担心出现任何问题。但如果我们要同时写一个公共的资源，那么以一个多重读独占写类型(TMultiReadExclusiveWriteSynchronizer，多读单写，也叫共享读独占写）的同步对象来说明其同步策略：

1、读取数据

（1）、BeginRead：检查是否正在进行写入操作，如果进行写入操作，则等待写入操作完成，以保证读取的数据是完整的，如果不等待，显然很可能会读到不完整的数据。如果没有，则进入共享读的状态。

（2）、线程读取数据内容，在此绝不能对公共资源进行写入。这是一个逻辑上的强制规则，但在编码实现却没有必然的约束，这就造成有些时候，错误的写入，会造成数据混乱。

（3）、EndRead：读取完成后，退出共享读状态。

2、写入数据

（1）、BeginWrite：检查是否有其它线程处于读或写的状态，如果处于该状态，则需要等待其它线程退出读写状态。无论其它线程是读还是在写，都不能进入写入状态，以避免破坏数据的完整性。一旦一个线程锁定了公共资源，其它线程就不能访问公共资源，必需同步等待这个线程对资源的锁定解除。

（2）、获得锁的线程修改公共资源内容。

（3）、EndWrite：写入完成后，退出独占写状态，锁被释放。

（4）、其它线程在获得执行时间片时，进行自己的读或写尝试。

理论上来说，多重读独占写的这种同步对象是最理想的锁，但由于其要求程序员在编码时必需注意到读和写两种不同的锁的区别，并保证相应的操作是没有问题的。而由于各种语法糖的引入，使程序员在开发过程中，有时候很难准确的满足这一要求。

临界（TCriticalSection）是另一个类型的同步对象，它与上面的多读单写类型的同步对象的区别是它不区分读和写操作，统一按照最糟糕的情况来考虑，也就是说，无论是读还是写，都同时只能有一个线程在执行，其它线程只能等待它

执行完成才能获取执行机会。其同步策略可以描述为：

1、请求进入临界（Enter）；

2、执行读或写代码；

3、离开临界（Leave）；

事件（TEvent）是另一个重要的同步对象，它只是一个信号。这个信号一旦被激活（SetEvent），那么直到重置（ResetEvent）前，每一个得到机会执行的线程都会退出等待状态。但事件有一个自动重置标志位，它在一个线程接收到信号退出等待状态后，就会自动重置为无信号状态，从而保证同时只有一个线程进入执行状态。事件实际上是一种通知，一般来说，其执行策略可以描述为：

1、需要等待某些数据就绪的线程进入等待状态（WaitFor）；

2、数据就绪后，触发信号（SetEvent）通知等待线程数据就绪；

3、等待中的线程处理数据，完成后重新进入等待状态；

还有多个同步对象类型，如互斥（TMutex）我们常用它来保证程序只运行一个实例，还有信号量（TSemaphore）、自旋锁（TSpinLock）、计数器（TCountdownEvent）等等，限于篇幅和时间，我们不再展开说明。

更多的关于同步对象和锁的信息，可以参考 MSDN 相关的文章，链接：https://msdn.microsoft.com/zh-cn/ms686360。

第十六章 附加选项

一般来说，我们不需要对 QWorker 中 Workers 这个包工头干涉太多，但如果你想干涉它的行为，QWorker 还是提供了一些额外的途径的。

1、控制工作者的数量

默认情况下， 包工头为了节省开支，会为雇佣的工人数量设置一个上限和下限。默认情况下，下限为 2，上限为逻辑 CPU 核心数量×2 +1，也就是说，对于一个四核处理器，默认的工作者上限是 2×4+1=9。在某些应用环境中，显然这过少了，所以，我们提供了两个属性来控制这些限制：

Workers.MinWorkers

这个属性用于控制工作者数量的下限，最小不能小于 1（你总不能让包工头自己应付临时来的少量工作吧）。

Workers.MaxWorkers

这个属性用于控制工作者数量的上限，最大值受系统资源的限制。Delphi / C++ Builder 中，每个线程栈的开销最低是 16KB，最大开销是 1MB，而一个 32 位程序的用户地址空间实际上只有 2GB 左右，所以，理论上最坏的情况下上限可以达到 2000 左右，但实际上你线程中肯定还牵涉到资源的分配利用，最坏情况下恐怕达不到这个数值。而且过多的线程对系统的资源调度能力也是一个很大的考验，一般个人不推荐太多。

2、长时间作业

一般情况下，作业都是简单快速执行的函数。如果一个作业需要很长时间才能完成，那么由此从事这个作业的工人将被长期占用，无法执行其它作业，这样显然会影响到其它需要处理的作业。QWorker 提供了长时间作业（LongtimeJob）选项，用于执行这类任务。对于长时间作业数量，QWorker 提供了一个上限为所有工作者数量的一半，超过这个数量时，长时间作业将无法投递执行。

从实际情况来说，长时间作业只是一个逻辑设定，而非硬性要求。你当然可以直接将需要较长时间才能完成的作业当作普通的作业投递并执行，但通过这样一个逻辑限定，可以帮助你重新思考你的业务处理逻辑，从而提升设计的品质。

从事长时间作业的工作者数量受属性 MaxLongtimeWorkers 限制，它的上限是工作者总数量的一半。

3、优化批量添加作业过程

正常投递作业的时候，QWorker 每投递一个作业，都会尝试去检查有没有空闲的工人能够处理这一作业。如果没有，就雇佣新的工人来处理作业，以保证作业得到快速处理。但在批量添加作业的情况下，这样的检查开销可以通过调用 DisableWorkers 来禁止检查工人是否空闲及获取新作业来提高投递的效率。

DisableWorkers 内部维护着一个计数器，每调用一次 DisableWorkers 这个计数器都会加 1，而与其配对的另一个函数 EnableWorkers 则负责将这个计数器减 1，当这个计数器减为 0 时，就会恢复正常的过程并启用必要的空闲工人来处理投递的作业。

您也可以直接设置 Enabled 属性的值来间接调用 DisableWorkers 和 EnableWorkers，但要注意一点，设置 Enabled 为 True 时，由于它是调用 EnableWorkers 来减少计数，所以未必在之后读取 Enabled 的值就是 True。

4、工作者被解雇的时间设置

QWorker 中，一个工人如果没有任何作业需要它去做，那么会引发一个计时，在这个计时完成后，如果它仍没有得到任何机会执行作业，它就会被解雇，占用的资源会被释放。这个计时由属性 FireTimeout 来指定，单位为毫秒，默认值为 15000，即 15 秒。

5、枚举工作者状态

如果我们的作业出现了死锁怎么办？ QWorker 提供了枚举工作者状态函数 EnumWorkerStatus 来列举出每一个工作者的状态和调用堆栈信息，帮助你快速定位错误来源。相关的内容请参考文章：QWorker 更新-在 Win32 平台上枚举作业状态加上堆栈显示。

6、其它一些辅助属性信息

Terminating 属性用来标志当前是否处于结束状态中，处于此状态时，Workers 对象自身正在释放中，不会接受任何新作业。

Workers 属性用来显示当前雇佣的工作者数量

BusyWorkers 属性用于显示当前忙碌工作者数量

IdleWorkers 属性用于显示当前空闲工作者数量

OutOfWorker 属性用于标记是否已经到达最大工作者数量（BusyCount==MaxWorkers）

NextRepeatJobTime 属性用于标记定时等重复作业最近的一次触发时间

OnCustomFreeData 事件用于处理用户自定义类型的作业附加数据的内存释放问题，具体可以参考文章：自定义作业数据指针释放方法

7、一些辅助函数的简单说明

QWorker 提供了一些简单的函数来辅助完成一些工作，下面是它们的简单说明：

MakeJobProc ：将一个全局函数转换为类的成员方法，这个主要是内部使用

GetCPUCount： 获取系统中 CPU 的核心数量，这个数量在 XE6 以后有一个 CPUCount 全局变量可以直接使用

GetTimestamp： 获取当前系统的时间戳，最高可精确到 0.1ms，但实际受操作系统限制

SetThreadCPU：设置线程运行的 CPU

AtomicAnd：原子位与操作

AtomicOr：原子位或操作

JobPoolCount：作业缓冲池中的已经缓冲的元素数量

JobPoolPrint：打印作业缓冲池中的对象信息

到此 QWorker 多线程编程的教程暂告一段落，如果有什么疑问，可以在群里或者在后面评论里咨询。在此，感谢大家的大力支持。